**NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:**
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

# CONVERSATIONAL PROGRAMMING--APL
# AN IMPLEMENTATION IN BLISS

A. J. Perlis
R. D. Fennell
F. J. Pollack
W. R. Price
M. F. Rizzo

Carnegie-Mellon University
Department of Computer Science
Pittsburgh, Pa. 15213

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research.
This document has been approved for public release and sale; its distribution is unlimited.

## ABSTRACT

As part of the ongoing research program in conversational programming an APL system has been implemented for the PDP-10. Since this system is to be a base for extensive study in conversational programming the system was programmed entirely in Bliss, a high-level programming language specifically designed for the writing of systems programs.

A few extensions to APL are included in this first version which supports both Teletype and IBM 2741/Datel terminals.

## WHY APL?

APL has been described in the book, A Programming Language, [5] by K. Iverson. It was intended to be a general data processing language but, because of its complicated notation and unwieldy alphabet, was little used even as a descriptive device until an interactive APL\360 version was implemented by IBM for the 360. The combined system and language have proved to be so enormously useful and successful in a wide range of applications, that implementations for a number of machines other than the 360 have been made, e.g., CDC 3600, 7600; XDS Sigma 7, Univac 1108, Burroughs 5500, IBM 1130.

The ARPA project at Carnegie-Mellon University developed a conversational language, LCC, derived from Algol for the IBM 360/67 TSS system. The decision to adapt LCC for the PDP-10 was an initial goal of the research program in conversational languages. However, it seemed more reasonable to use APL as a base for extensions since APL's computational component is more powerful than LCC. Put another way, APL is a more powerful language than Algol in many important respects, not the least of which is conciseness.

Like many other excellent programming languages, APL suffers from an omission of important functions both in its computational and system aspects. Some of these were already in LCC, others were to be provided in a next extended version. Consequently it seemed quite reasonable to construct an APL system as a base for conversational programming research.

APL, as an array processing language, raises a number of interesting and important optimization problems that, while still present, do not surface so naturally in scalar processing languages like Algol and FORTRAN. Basically these problems deal with the scheduling associated with the partial processing of partial arrays.

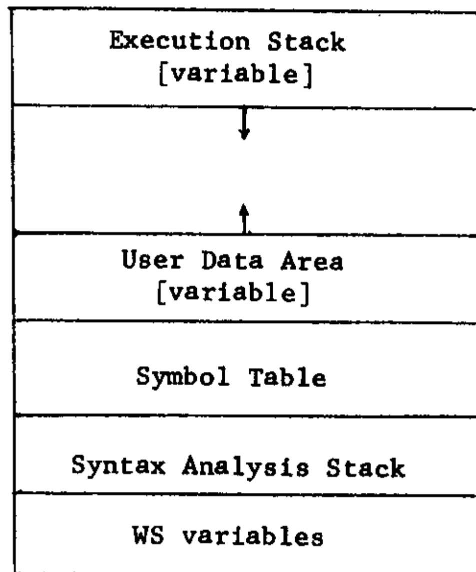
Furthermore, APL permits any identifier to denote data of varying size and shape; and this, coupled with the dynamic modification of program text, raises a number of interesting compilation problems which require solution since many conversational programs ultimately require compilation into efficient machine codes.

Since the APL system is to be an evolving one it is critical that it be coded in a powerful statement language so that the code may be easily produced, understood and altered. Fortunately the BLISS language [7] developed at Carnegie-Mellon University satisfied all of these requirements. Furthermore, its development was being completed when the APL effort commenced so that the APL system could be used to test BLISS's claim to be a system building language.

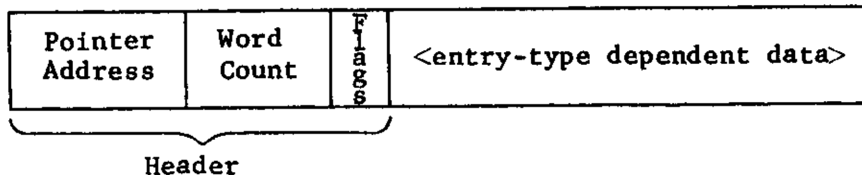
## SYSTEM LAYOUT

From the user's point of view, APL\10 resembles very closely IBM's APL\360 as described in the APL\360 Reference Manual by Sandra Pakin [6] and the APL\360: User's Manual [4]. A subsequent section of this paper describes the extensions and modifications incorporated into APL\10.

The basic organizational unit in the APL\10 system is the workspace, which occupies the major portion of each user's PDP-10 low segment (with the sharable APL interpreter occupying the high segment). The workspace may be depicted as follows:



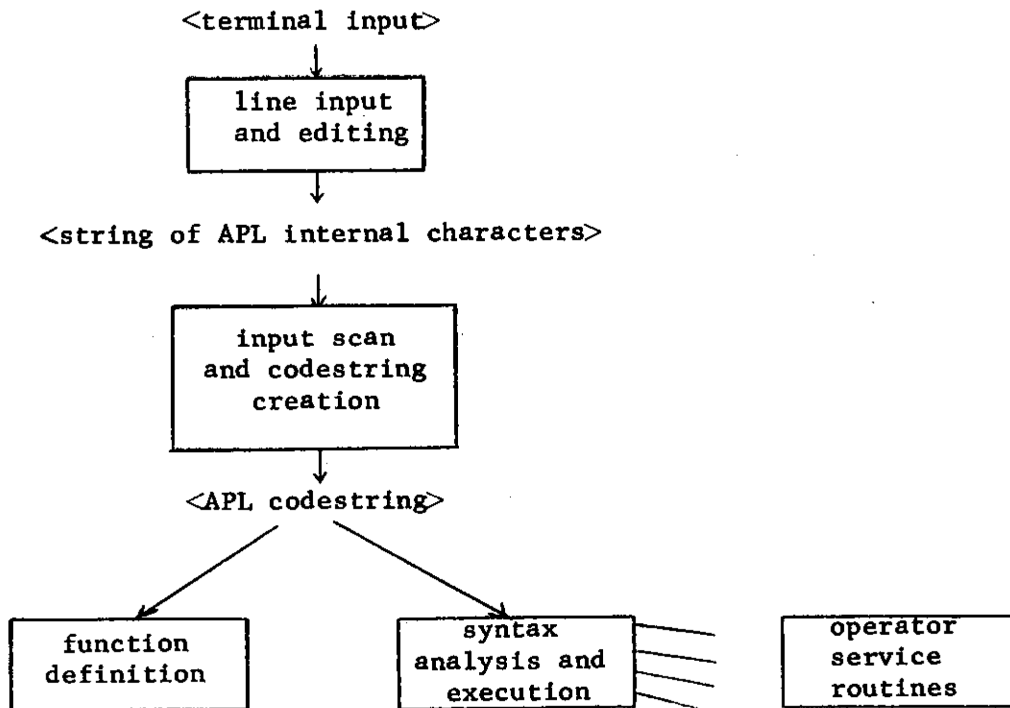

The execution stack contains function activation records, syntax units, and associated data. The user data area contains user-defined data items, such as function definitions and variable values. The general format of a data entry is:




The 'flags' tell whether or not the data entry is a list-type entry (see the section on Function Definition) and whether or not the data entry is garbage (see the section on Garbage Collection).

The fixed-size symbol table is discussed in more detail below. Syntax analysis is essentially done by the Conway Transition Diagram method [2], and the syntax analysis stack is used to maintain a record of the syntax analysis path being traced. The workspace-variables area contains various pointers and workspace constants (such as the number of significant digits to be used during display, the width of the printed page, and the index origin).

The basic interpretation sequence for APL\10 may be broken into several distinguishable actions:



LINE INPUT AND EDITING

The basic function of input line editing is to accept an ASCII input line from a teletype or Datel/2741 (the current version of APL\10 uses the input conventions for 2741-like terminals established by PDP-10 monitor modifications introduced at Carnegie-Mellon University), and edit that line so as to create a string of internal APL characters which reflect the physical appearance of the line as typed at the user's terminal. This principle of visual fidelity entails editing out any backspaces that may have been input, creating a single internal overstrike character wherever two characters have been overstruck at the terminal, translating keyword or escape mode inputs (for teletypes) to their corresponding single APL characters, etc. Illegal overstrikes must be detected so that the user can be prompted to enter a correction line.

Due to the extensive character set required by APL, the internal APL characters are 9-bit bytes. Actually, only 8-bits are required, but 9-bit bytes allow for an even division of the PDP-10 36-bit word and allow for adequate future expansion of the internal codes required (such as new overstrikes).

A similar translation process takes place upon outputting an APL string of 9-bit characters. Translation tables are required for converting single printable characters to their ASCII equivalent for expanding overstrikes to an expanded ASCII form (whether it be <char> <BS> <char> for a Datel or a keyword or escape mode representation for a teletype). Teletypes also require that many APL special characters be expanded, due to the limitations in the teletype character set.

INPUT SCAN AND CODESTRING CREATION

An edited 9-bit string is then passed to a lexical analyzer to be scanned and transformed to a form which is readily acceptable by the syntax analyzer. This intermediate form is known as a codestring and is essentially a one-to-one mapping from the edited 9-bit string. Negligible syntax checking is done during codestring creation; and once the codestring is produced, the 9-bit source line is discarded. Thus only one internal representation of an APL statement is retained -- the codestring. From the codestring it is easy to reproduce the corresponding source line and also the codestring is readily parsed by the syntax analyzer due to APL's simple right-to-left evaluation scheme and the lack of hierarchical ordering among the operators. Syntax analysis can be performed quite readily on the simple intermediate representation of the codestring.

Before discussing the format of the codestring, the symbol table structure should be explained. The APL\10 symbol table is of fixed size, consisting of a number of two-word entries (STE's). All references to a variable name are made via the symbol table. The STE location is initially established by hashing on the first 12 characters of the given identifier, then using a quadratic search technique to resolve any conflicts. The format of an STE is:

| absolute addr. of value data entry | type | print-name length | first char. if short | rest of printname if short (≤ 5 chars) |
|---|---|---|---|---|

| absolute addr. of printname data entry if > 5 char | |
|---|---|

The value data entry address field is set by the syntax analyzer whenever a value is associated with an identifier. The type field indicates whether the associated item is a variable, function definition, group, etc. The printname of the identifier is stored in the STE if it is less than 6 characters; otherwise it is put in a special printname data entry which is of the form:

| absolute addr. of STE | word count of this data entry | 9-bit chars of printname, packed 4 to a word |
|---|---|---|

Entries in the symbol table are never deleted once they are created (although they may become 'undefined' by 'erasing' a variable name); when a variable is given a new value, the old STE is re-activated if its type had been 'undefined'. The reason for not deleting entries is that there may be arbitrarily many pointers to a STE from codestrings in the workspace (recall, all references to a variable are made via the symbol table); and finding all such occurrences would be impractical. So the STE type field is merely marked as 'undefined'; and when those codestrings are executed, any references to an undefined datum will generate a 'value error'. Note: A 'copy' operation may be used to clear a symbol table of unnecessary identifiers by saving the active workspace, clearing the workspace, then 'copying' that saved workspace. The copy operation defines only those entries which possess values.

Now to discuss the actual codestring creation. As mentioned before, the codestring is essentially produced by a one-to-one mapping of the

source string. The various entities handled by the lexical analysis phase are:

1. Identifiers are transformed to an 18-bit offset from a symbol table base, and the corresponding STE is created.

2. Numbers are translated to a PDP-10 internal format which distinguishes between bit vectors, integer vectors, and single precision floating point vectors.

3. Colons are checked as label delimiters.

4. Quoted strings are transformed to a 9-bit character vector format.

5. ∇'s are used to delimit the function definition phase (described below).

6. Carriage returns are used to trigger end of statement processing (setting the header of the codestring data entry, calling the syntax analyzer for immediate statements, inserting the codestring in a function definition directory for non-immediate statements, etc.).

7. Blanks are ignored in codestring creation, except as delimiters (as of identifiers and numbers) and when they occur in quoted strings.

8. Special characters (operators) are simply placed into the codestring according to their 9-bit representation.

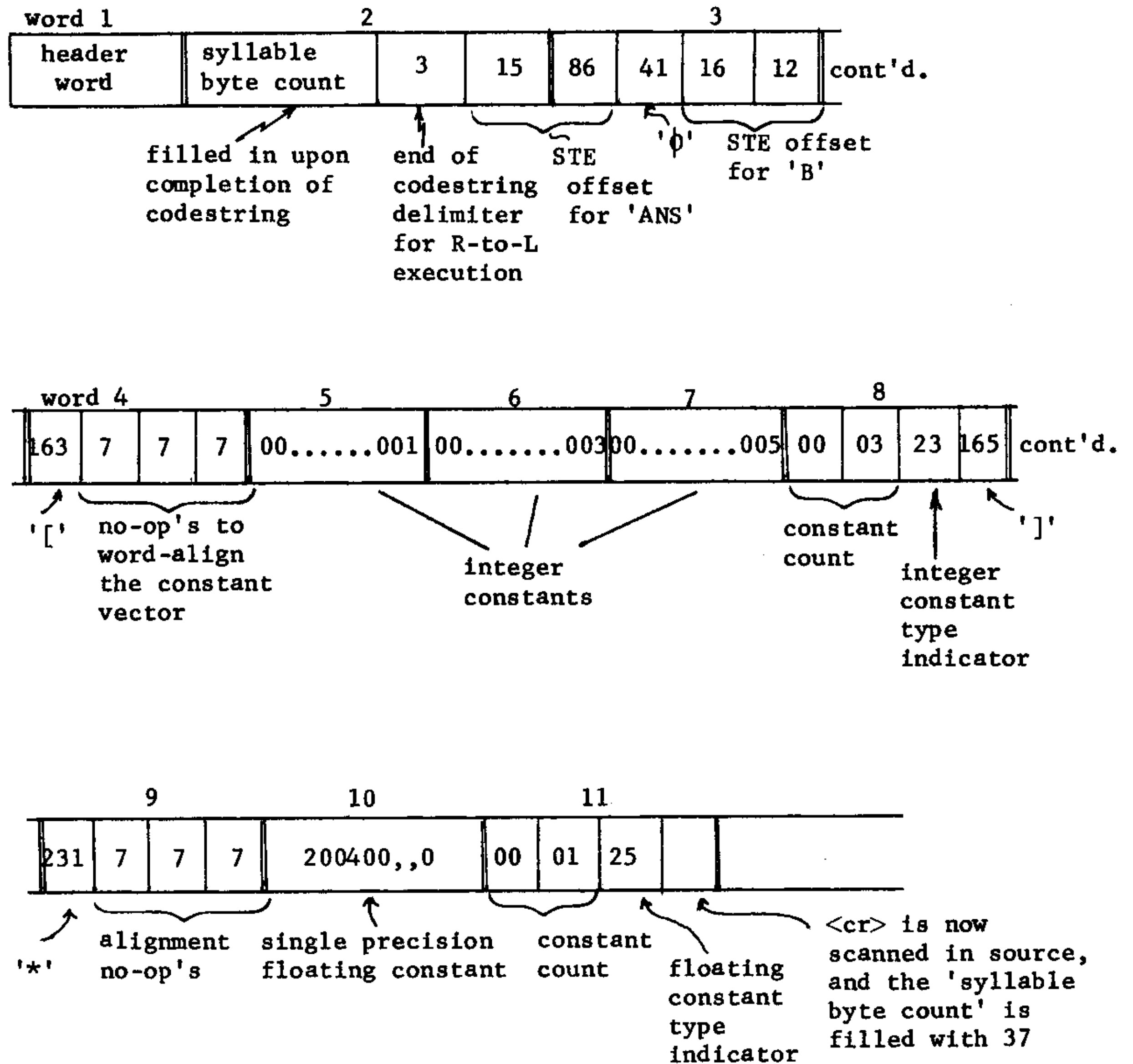
Codestrings are comprised of a sequence of syllables. Syllables are of two types: long syllables of 18 bits (these include symbol table offsets and various counter fields for vectors) and short syllables of 9 bits (these include the special characters and various type indicators). The syllable types are distinguished by the rightmost bit (thus, short syllables are stored as 1 + 2 x value). Note that the symbol table offsets are always even since STE's are two words in length.

Scalars and vectors occurring in the input line are put into the codestring as a sequence of values (fullwords for integers and floating point constants, and bits for a vector of only 0's and 1's) followed by a constant count (long syllable) and the vector type (short syllable). This trailing information facilitates the right-to-left syntax scan.

As an example, consider the following input line:

ANS ← B ⌽ C[1 3 5] * 0.5 <cr> <lf>

The corresponding codestring is:

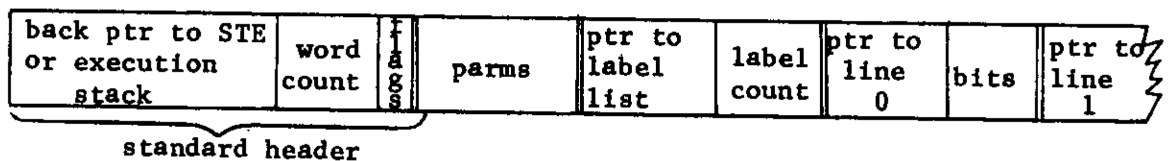



The codestring is created starting at the first available location in the user data area of the workspace (that is, on top of the user data area stack).

FUNCTION DEFINITION

Once the codestring has been generated, it may be executed by calling the syntax analyzer (for an immediate statement) which can thence generate output and/or create value data entries to store the results of the calculation. If the codestring was created in function definition mode (as opposed to immediate mode), then instead of calling the syntax analyzer to execute the codestring, the codestring is added to the directory of the currently open function definition. Recall that one enters function definition mode by scanning a '∇' during codestring creation. Scanning a second '∇' causes one to leave function definition mode and return to immediate mode, after attending to the details of properly closing the function definition.

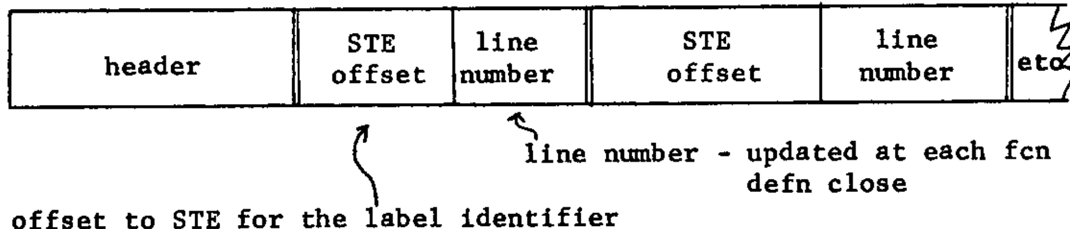
The actual function definition makes use of another form of data entry: the list data entry. As it pertains to functions, the list data entry is essentially header information, which characterizes the particular function, followed by a series of one word entries which describe each line of the function. More specifically, the format of a function list data entry is:




In 'flags', the 'list-type' bit is set on. The 'parms' word contains information concerning whether or not the function is locked, and how many

local variables, parameters, and lines the function has. The 'bits' field of each line entry contains information on whether or not the particular line is labeled or is to be traced or is to be stopped.

Notice the third word in the function directory contains a pointer to the label list. Line labels in APL\10 are treated as variables local to the execution of a given function rather than as variables global to the entire workspace. Thus, upon each execution instance of a function, the syntax analyzer retrieves the current label values from the list of line labels and treats these variables just as any other local variable in the function. The format of the label list is the following:




SYNTAX ANALYSIS AND EXECUTION

The syntax analyzer operates directly on the codestrings generated by the lexical analyzer and controls the interpretation. Since there is no operator precedence, APL statements are interpreted from right to left. The syntax analyzer uses Conway transition diagrams of four basic types: (1) a statement diagram, (2) a list diagram, (3) an expression diagram and (4) a basic diagram. A diagram is made up of paths, and each path contains information as to what type of element in the codestring to expect and what actions are to be performed if a match is or is not made. A fixed size diagram stack is used in this syntax analysis.

An attempt was made to implement some of Abrams' [1] ideas of beating and dragging of APL expression. *Dragging* is the process of delaying execution of APL operators, thereby reducing the number of temporary locations in the evaluation of an APL expression. *Beating* is the process of operating on a descriptor of an array rather than on the array itself. The basic form of a descriptor is seen in Figure I. When the syntax analyzer scans a variable or constant, a descriptor is created and put on the execution stack. Consider the simple APL expression: D ← C + A + B. First a descriptor is made for B, then a '+' is put on the stack, then a descriptor for A is made. Since scalar operators can be delayed, a descriptor for A + B is made, but the actual addition is not yet performed. The descriptor for A + B looks like a descriptor for a variable except that the data-entry pointer is replaced by a '+' and OP1 contains a pointer to the descriptor of A and OP2 contains a pointer to the descriptor of B. Next, a '+' is put in the stack and a descriptor is made for C. Then a descriptor is made for C + (A + B). Up to this point no additions have been performed. The assignment operator requires the value of the expression, so the expression is now evaluated. If A, B, and C were 100 by 100 matrices, the traditional method would have allocated 10,000 locations to store A + B, executed 10,000 stores of A + B, and 10,000 loads of A + B. By delaying, those locations and operations are not needed.

To illustrate dragging consider the APL expression D ← A + ⍉B. Instead of doing the actual transpose of B, only the descriptor of B is changed. If B *is* 40 50, then the R-VECTOR *is* 40 50 and the DEL-VECTOR

FIGURE I

| DATA-ENTRY POINTER | FLAGS |
|---|---|
| RANK | ABASE |
| OP1 | OP2 |
| R-VECTOR[1]<br>R-VECTOR[2]<br>⋮<br>R-VECTOR[RANK] | DEL-VECTOR[1]<br>DEL-VECTOR[2]<br>⋮<br>DEL-VECTOR[RANK] |

FOR AN IDENTIFIER, CONSTANT, OR J-VECTOR, OP1 ≡ OP2 ≡ ∅
IF A ← 3 4 5 ρ∅, THEN
R-VECTOR = 3 4 5
DEL-VECTOR = 20 5 1
RANK = 3

TO GET ADDRESS OF A[2; 3; 4]
ADDR = DATA-ENTRY POINTER + ABASE + (2×20) + (3×5) + (4×1)

is 50 1. The transpose operator will change the R-VECTOR to 50 40 and the DEL-VECTOR to 1 50. This process, therefore, saves several operations and temporaries. See Figure II for other operators that are beaten.

The concept of a J-vector has also been implemented. A J-vector is a sequence of integers such that the difference between two successive integers is 1 or ¯1. Therefore, a J-vector can be described by specifying a base, length, and direction. The descriptor of a J-vector looks like a descriptor of an identifier except that it has no data-entry pointer, RANK=1, RVEC=LENGTH, ABASE=BASE, DEL=1 or ¯1. The only way to create a J-vector is by using the monadic iota(ι). Since the only space a J-vector takes is the space needed for the descriptor, it is possible to evaluate +/ι500000 when the workspace is, e.g., only 8K words.

The main advantage to doing beating and dragging is to execute operations on large arrays faster at the expense of small arrays. That is, it is better to execute a two minute program in one minute even if it means executing a 0.1 second program in 0.2 seconds. To be consistent with this idea, most operators cause machine code to be generated, executed, and then thrown away. This has to be done since in APL sizes and shapes of arrays are so dynamic.

To the APL user, it appears there are only two types of data: character and numeric. However, internally there are four types: boolean (1 bit), integer (36 bits), floating point (36 bits), and character (9 bit). Beating, dragging, and code-generation make it difficult to maintain this transparency. Consider the following expression:

FIGURE II

FROM ABRAMS' THESIS [1]

OPERATORS THAT ARE BEATEN

```
Q↑M (TAKE)
     ABASE ← ABASE + DEL +. × (Q < 0) × RVEC - |Q
     RVEC ← |Q

Q↓M (DROP)
     ABASE ← ABASE + DEL +. × (Q > 0) × |Q
     RVEC ← RVEC - |Q

⌽[J]M (REVERSAL)
     ABASE ← ABASE + DEL[J] × (RVEC[J] - 1)
     DEL[J] ← - DEL[J]

⍉M (MONADIC TRANSPOSE)
     RVEC[0 ¯1 + ⍴⍴M] ← RVEC[¯1 0 + ⍴⍴M]
     DEL[0 ¯1 +⍴⍴M] ← RVEC[¯1 0 + ⍴⍴M]

A⍉M (DYADIC TRANSPOSE)
     R ← RVEC
     D ← DEL
     RANK ← (⌈/A)
     I ← 1
     DEL ← RANK ↑ DEL
     RVEC ← RANK ↑ RVEC
     REPEAT: RVEC[I] ← ⌊/(I=A)/R
     DEL[I] ← +/(I=A)/D
     → REPEAT × ⍳RANK ≥ I ← I+1

M[[J]SCALAR] (subscripting with a scalar in J^th coordinate)
     ABASE ← ABASE + DEL[J] × SCALAR - 1
     DEL ← (J ≠ ⍳RANK)/DEL
     RVEC ← (J ≠ ⍳RANK)/RVEC
     RANK ← RANK - 1

M[K]J LEN, ORG, S] (subscripting with a J-vector in K^th coordinate.
                    J 3, ¯2, 1 is ¯2, ¯1, 0.  J 4, 2, ¯1 is 2, 1, 0, ¯1).
     ABASE ← ABASE + DEL[K] × (¯1) + ORG - (LEN-1) × S = -1
     RVEC[K] ← LEN
     DEL[K] ← DEL[K] × S
```

A ← 2 * 2 x (20 4 3). Integer code is produced for this expression; on first iteration A[3] is calculated to be 64, on second iteration A[2] is found to be 256, but the third iteration causes a fixed-point overflow (2 * 40). Therefore, previous results are thrown away and new code must be generated to do the operation in floating point.

## IMPLEMENTED OPERATOR EXTENSIONS

So far two operators have been extended and one added. The decode and encode operators have been extended to arrays.

A is a scalar or vector.

B is a scalar or an array.

k is a scalar or 1-element array

R ← A [k] B (DECODE)

if '[k]' is omitted [ρρB] is assumed

ρR ≡ (k ≠ ιρρB)/ ρB

R ← A⊤B (ENCODE)

ρA

ρR ≡ (ρB), ρA

Therefore, if A is a vector, B ≡ A⊥A⊤B

The Subscan operator (\) has been implemented and is defined as follows:

R ← ⊙\[k]B

if [k] is omitted [ρρB] is assumed

⊙ is any logical or arithmetic scalar operator

ρR ≡ ρB

Suppose (ρρB) ≡ 3 and k ≡ 2, then

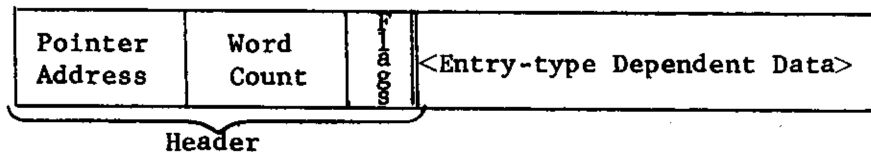
R[;J;] ≡ ⊙\[2](J ≤ ι(ρB)[2])/[2]B

Hence, -\ι3 ≡ 2 ¯1 3

Note that -/ι3 ≡ 2. If subscan were done forward, -\ι3 would equal 1 ¯1 ¯4, none of which equals the value of the reduction. Subscan does not work on relational operators, because of the following problem: should ≥\5 6 be defined as 0 6 as the above definition implies or perhaps 0 1 (1 ≡ 6 ≥ 6)?

GARBAGE COLLECTION

As described above, the basic information structure used to retain a user's input and execution results in the data entry. Data entries come in various formats (for example, codestring data entries, printname data entries, value data entries, and function list data entries), but each type has a standard header word of the following format:




The 'pointer address' is an 18-bit pointer to the symbol table entry (for value data entries and function list data entries) or to the second word of the STE (for printname data entries) or to the appropriate member of a function list data entry (for codestring data entries which belong to a function). Thus each data entry points somewhere and that somewhere points back to the data entry. Hence, for example, since all references to a data value are made via the STE for the particular identifier, the value may be reassigned by creating a data entry for the new value, pointing that data entry to the STE, pointing the STE back to the new

entry, and marking the old value data entry as garbage. Notice that no reference made to the variable from a codestring had to be changed. The 'word count' field of a data entry gives the length of the data entry. As mentioned previously, 'flags' tells whether or not the data entry is a list-type entry and whether or not the data entry has been marked as garbage.

As a user session progresses and the user edits existing functions, replacing and deleting function lines, and repeatedly assigns value data entries to identifiers (via immediate statements and function execution), the discarded data entries are marked as garbage. Garbage entries (those with the garbage-bit in the 'flags' field of the data entry turned on) include old value data entries, replaced and deleted function lines (codestring data entries), executed immediate lines (codestring data entries), and old function and label directories (new ones are created each time a function is closed). To reclaim the space occupied by these garbage entries, a garbage collection routine is activated at appropriate times (such as at the end of each executed line and at each function definition closing).

Garbage collection is accomplished by using the header information contained in each data entry to search through the user data area of the workspace for non-garbage items and to compact those items over the entries to be reclaimed. Compaction is complete when the top of the old user data area stack is reached; the stack pointer is then reset to the new stack top.

The actual compaction process involves moving the good data entries via a block transfer and resetting the back pointer of the item to which

the data entry points so as to point to the new location of the data entry. Notice that since all codestring references to variables are made via the symbol table (which does not move), no reassignment of any STE offset pointers which may occur within a codestring is necessary when a codestring is moved. Relocation with respect to a list data entry such as a function directory requires special attention. In addition to revising the back pointer to the directory header, the back pointers contained within the headers of each function line (codestring data entries) must be revised to reflect the new position of the function directory entry. This special case is detected by checking the list-bit of the 'flags' field of each data entry (once it is determined that the data entry is to be relocated). The corresponding case of garbaging a list-type data entry (as for example in erasing a function) is handled similarly: if the list-bit is on, each member of the list must also be marked as garbage.

This garbage collection process is quite efficient and the time required is not noticeable to the user. Since garbage collection is done relatively often, the amount of relocation necessary is relatively small. Also, since non-garbage items tend to migrate toward the bottom of the stack and most of the relocation occurs in the more active top section of the stack, a pointer is maintained to mark the lowest piece of garbage so as to avoid rescanning all the more permanent items at the bottom of the stack. Thus the garbage collection process usually needs not scan through all of the data entries in the user data area.

## APL\10 INTERPRETER OVERVIEW

The following pseudo-Bliss code illustrates the major controlling loops of the APL\10 interpreter.

```
begin
   <external Macro-10 table names>;         %tables for character translation,
                                             error messages, etc.%

   <declare global routines>;               %these include service routines,
                                             I/O and line-editing routines,
                                             'LEXICAL', 'INIT', and 'SOPROC'%

   <declare global variables>;              %I/O buffers, global pointers,
                                             system globals, and global flags%

   <declare workspace>;                     %ws variables, syntax analysis
                                             stack, symbol table, initial
                                             data area/execution stack space%

   INIT( );                                 %clear some system globals; open
                                             I/O channels%

   SOPROC( );                               %accept first input line and allow
                                             sign-on; load continue-workspace,
                                             if necessary%

   while 1 do
      begin
         if <more Bliss-stack space is needed for a workspace size adjustment>
         then <revise the Bliss-stack pointer upward>;

         <perform any load/copy/size request, including any necessary intra-
          workspace relocation>;

         if <less Bliss-stack space is needed>
         then <adjust the Bliss-stack pointer downward>;

         LEXICAL ( )                        %the major interpreter routine%
      end
end;                             %end of APL interpreter%
```

The major interpreter routine, 'LEXICAL', may be outlined as follows:

```
global routine LEXICAL =
   begin
      <declare service routines>;
      <declare 'SCANNER' with its included lexical routines (one of which
                calls 'SYNTAX' to accomplish codestring execution)>;

      while 1 do
         begin
            <reset pointers and status bits>;
            <accept and edit an input line>;
            <handle any function line editing or function display requests>;

            if ')'                          %system command?%
            then if <processing of system command requires a workspace
                       size change>
                 then return                %size changes handled in outer loop%
                 else <process command>
            else SCANNER ( )                %call 'SCANNER' to scan input string,
                                             create a codestring, and either call
                                             'SYNTAX' for codestring execution or
                                             enter the line in a function directory%
         end
   end;                   %end of major interpreter routine%
```

## THE PDP-10 IMPLEMENTATION

To the PDP-10 Monitor, APL users are no different than any other user. Each APL user runs under the PDP-10 monitor as one job. The APL system is a two-segment program. The high segment is normally a pure write-protected sharable segment and is presently 37K words in size. The impure non-shared low segment normally is 1K + the size of the active workspace. Currently because of the communications commands the system is restricted to 36 users.

Two modifications to the standard DEC Monitor were required. The first modification is a relatively minor alteration to the ENTER UUO to allow a user to create a disk file on another user's UFD with the protection specified in the parameter block associated with the UUO. The second modification is a more complex addition to the monitor to allow for the use of IBM 2741-like terminals with a special APL mode. The latter modification is not necessary if Model 33 Teletypes will suffice.

### RUNTIME ENVIRONMENT

The shared high segment of the 2-segment APL system consists of: (1) sharable pure code, (2) read only data, and (3) read-write communication data. Under normal circumstances the high segment is write-protected. The only time the write protection is off is when a user updates communication data. (The hardware write protect feature is off only for the user doing the updating and remains in effect for all other users.)

The read-only data consists of translation tables used to convert the ASCII input characters into an internal code and vice-versa and to convert any overstruck character into a single internal character and vice-versa, message texts, command table, and code templates.

The command table contains the first four letters of each system command along with information regarding what parameters to expect in the input line and what routine to call to execute the command. In execution of APL statements the code templates are pieced together to create a portion of code to which control is eventually passed and results in the execution of the APL statement.

The read-write communication area consists of the port buffers and a message pool. There is a buffer associated with each of a possible 36 ports. Each buffer contains the APL number and identification of the user associated with the port and flags indicating whether the user is privileged and whether there is a message for the user. The information in the buffers is needed to implement the )PORTS command.

The message pool is a block of core reserved for storing messages among users. The sender takes a chunk from the pool into which the message is placed and then sets the bit in the receiver's port buffer indicating that a message is pending. Prior to each request for input from the terminal, APL checks the bit to see if the user has a message. If so, the bit is turned off, all messages are printed at the terminal, and the space taken is returned to the message pool if no one else is to receive the message. There is a mutual exclusion variable which prohibits any simultaneous updating of the shared read-write data area.

The APL system is implemented in the implementation language Bliss. Bliss is an Algol-like language employing a runtime stack. One of the goals of the APL system was to provide a variable workspace size which presented a problem in positioning the workspace and the Bliss runtime stack. Making the impure low segment large enough for a Bliss stack and the largest workspace would defeat the purpose of the variable workspace size. Positioning the Bliss stack followed by the workspace (in order of increasing address) would be a bad practice because on block entry Bliss space is reserved by adding the number of locals to both halves of the stack register. The danger is that the hardware can only catch stack overflow by push(j)-pop(j) instructions when the left half of the accumulator goes to 0 or -1 respectively, a condition which may be bypassed by the addition of a constant to both halves of the stack register. The inability to catch stack overflow could result in damage to the workspace when the stack overflowed into the workspace.

Another possibility is the opposite of the above: position the workspace followed by the Bliss stack and relocate the Bliss stack upward or downward as the size of the workspace increases or decreases. However, the relocation of the Bliss stack is virtually impossible since the relocation process would have to be able to determine which of the stacked values are addresses.

The solution to the dilemma was achieved by observing that the last mentioned solution could be effected if the Bliss stack were empty at the time it had to be moved. Thus, whenever it is necessary to reposition the Bliss stack, all Bliss routines and blocks containing local variables

are exited to an environment where the only variables referenced are global (variables bound to a fixed address). Unfortunately this scheme in the APL system restricts the relocation of the Bliss stack to sign-on, )LOAD, )CLEAR, )SIZE, or )COPY commands. Since the Bliss stack is far from empty during function definition and statement execution the size of the workspace cannot be varied completely automatically.

Normally the low segment of the APL system consists of I/O buffers, global state variables, the active workspace, and the Bliss runtime stack in order of increasing addresses (Figure III).

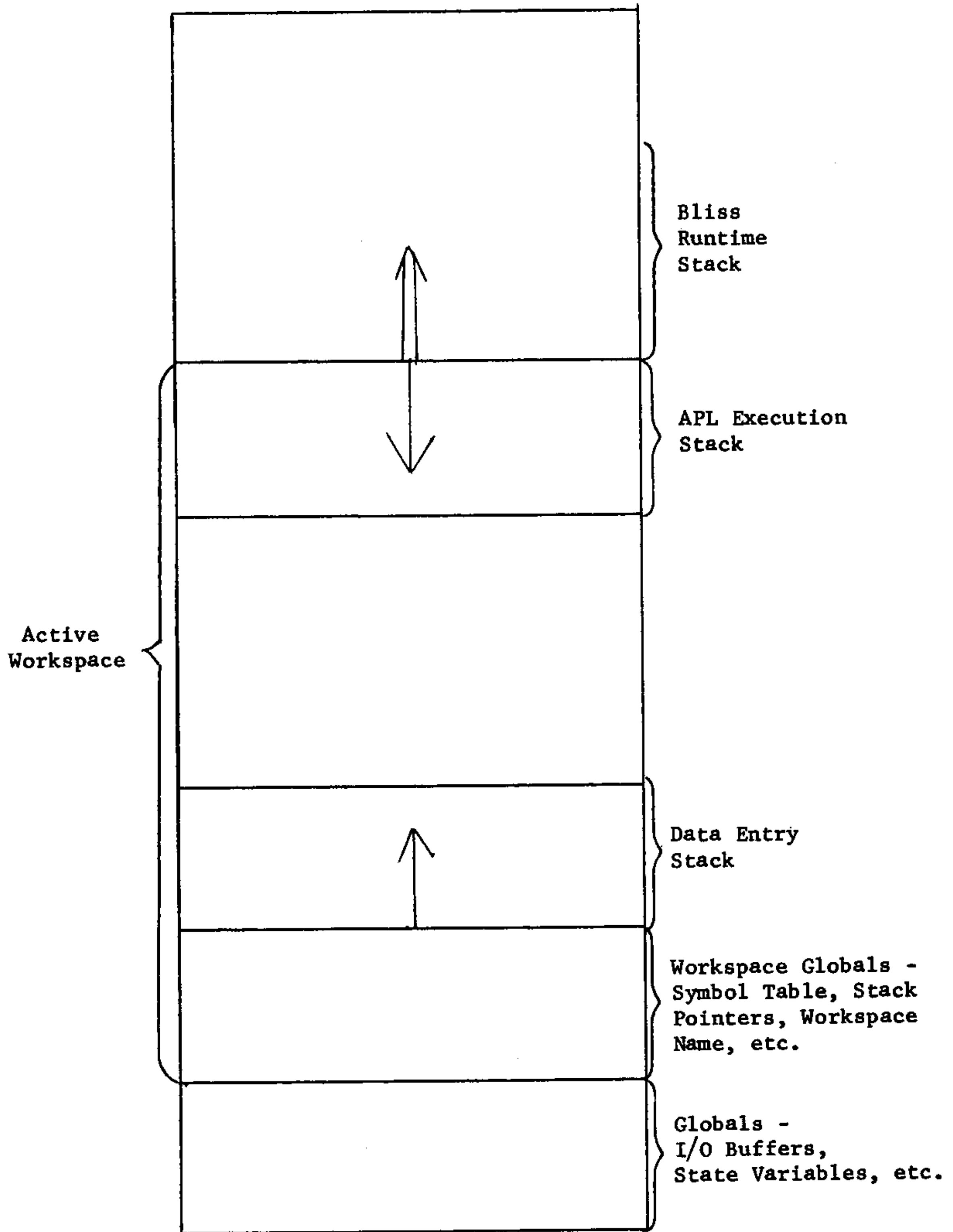
Initially the Bliss stack base (Figure IV) is positioned at the normal base of the workspace. Once the user has satisfactorily completed sign-on, the system establishes an active workspace and moves the Bliss stack base above the top of the workspace.

Use of the )COPY command requires the copy workspace be loaded into core. To accomplish this the APL system exits to the environment of an empty Bliss stack, loads the copy workspace just above the active workspace, and establishes the Bliss stack above the copy workspace (Figure V). Routines are then called to effect the copy. When the copy is complete, the system exits to the environment of an empty Bliss stack and resets the stack base to just above the active workspace.

WORKSPACES

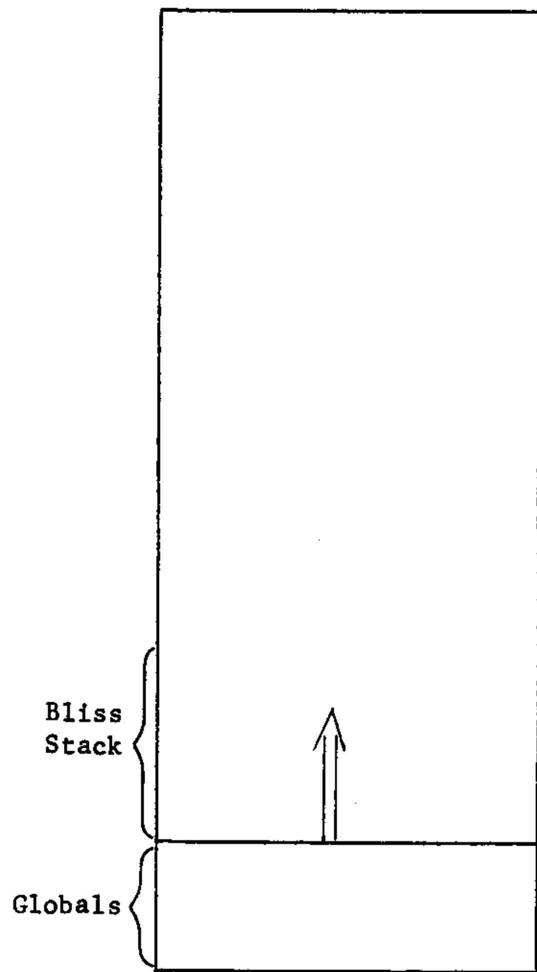
One of the goals of this effort was to have the PDP-10 APL appear to the user to be identical to APL\360. The library system of APL\360, where users can access public workspaces and the private workspaces of

FIGURE III

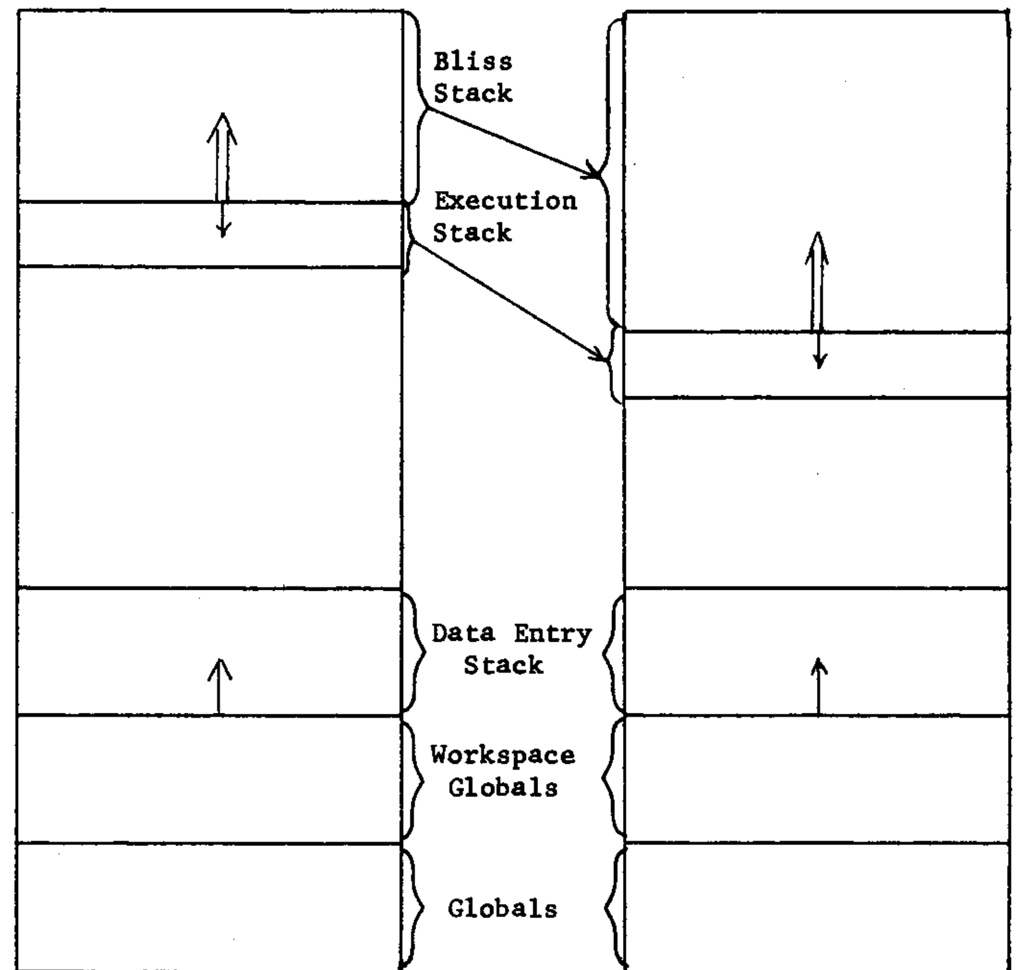



NORMAL LOW SEGMENT
CONFIGURATION

Figure IV

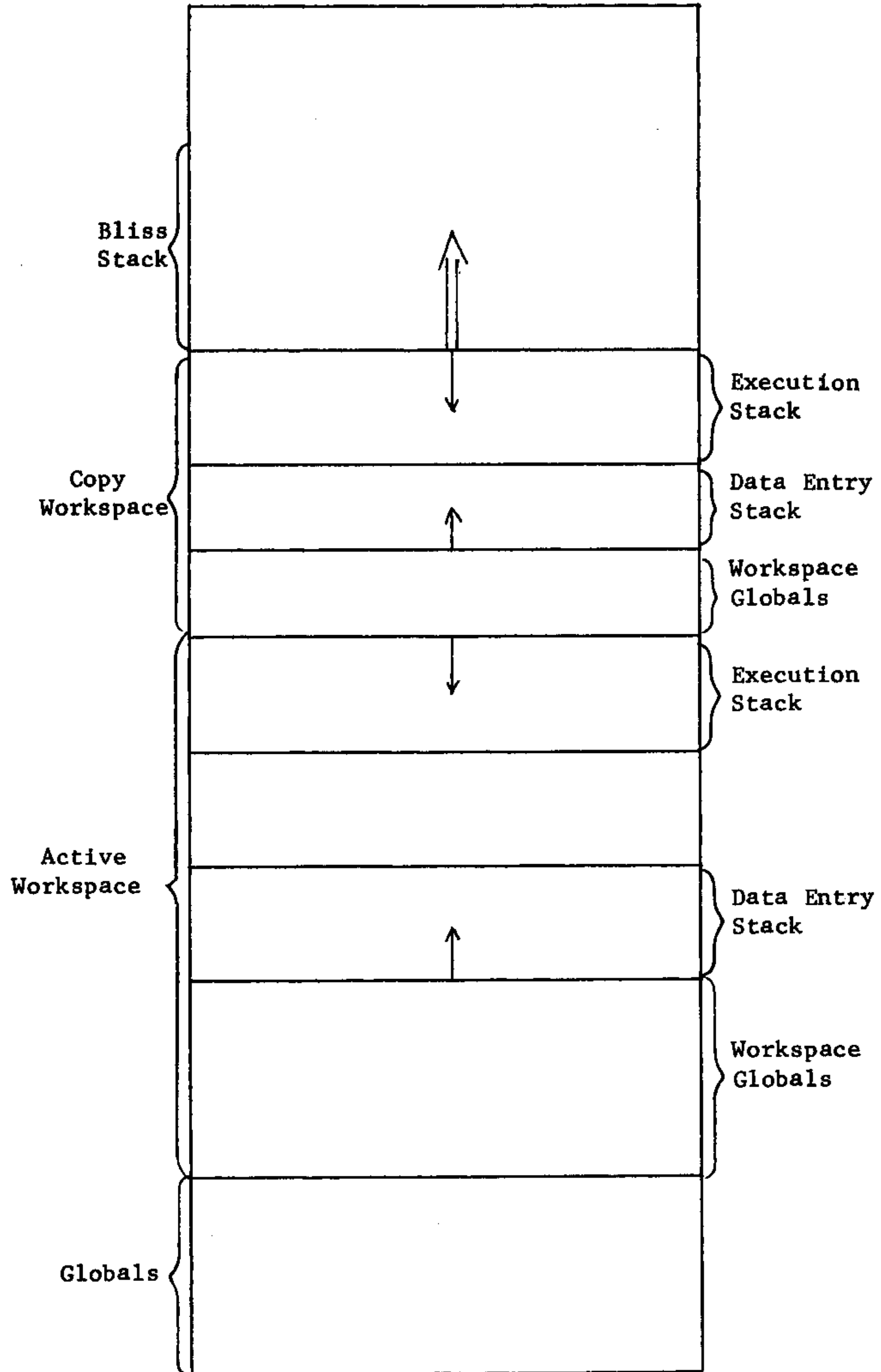



LOW SEGMENT DURING
SIGN ON




CHANGE IN RUNTIME ENVIRONMENT
RESULTING FROM SIZE, LOAD, AND CLEAR COMMANDS

FIGURE V




LOW SEGMENT CONFIGURATION
DURING COPY

other users, required that workspaces be stored in file space of one PDP-10 user. For each APL user there is a PDP-10 disk file that acts as a directory for the user's workspaces. This file is called a <u>user file</u>. The PDP-10 name of the user file can be determined uniquely as a function of the APL number.

The user file (Figure VI) is one disk block, and contains the user's identity (a project-programmer pair), the sign-on password, the disk and workspace quota, disk and workspace used, the cumulative connect and CPU times, 20 workspace buffers, and flags indicating whether the user is privileged, locked out of the system, currently signed-on, and whether a continue workspace should be loaded the next time the user signs-on. Although disk space is the primary resource upon which quotas should be established, a quota upon the number of workspaces a user is allowed to have was also established to discourage users from creating a new workspace for each function. (Eleven disk blocks are required to save a clear workspace because of the symbol table.)

The workspace buffer provides a means of mapping a long workspace name into a shorter PDP-10 file name. Each workspace buffer contains the name of a workspace, the password, if any, needed to load the workspace, and counts of the number of disk blocks for the data entries and the number of disk blocks for the APL execution stack. When workspaces are saved, only the meaningful parts, the low data entries and the execution stack are saved. The pool between the two stacks is not saved. When a workspace is saved, the user file of the library number under which it is to be saved is accessed. The user file is then searched for a workspace

FIGURE VI

<table>
<tr><td colspan="8">ID</td></tr>
<tr><td colspan="8">SIGN-ON PASSWORD</td></tr>
<tr><td colspan="8">CUMULATIVE CONNECT TIME</td></tr>
<tr><td colspan="8">CUMULATIVE CPU TIME</td></tr>
<tr><td>DISK QUOTA</td><td>DISK USED</td><td>WORKSPACE QUOTA</td><td>WORKSPACE USED</td><td>PRIVILEGE BIT</td><td>LOCK BIT</td><td>SIGNED ON BIT</td><td>CONTINUE BIT</td></tr>
<tr><td colspan="8">WORKSPACE NAME</td></tr>
<tr><td colspan="8">PASSWORD</td></tr>
<tr><td colspan="4">LOW WORKSPACE SIZE</td><td colspan="4">HIGH WORKSPACE SIZE</td></tr>
<tr><td colspan="8"></td></tr>
<tr><td colspan="8"></td></tr>
</table>

WORKSPACE BUFFER #1

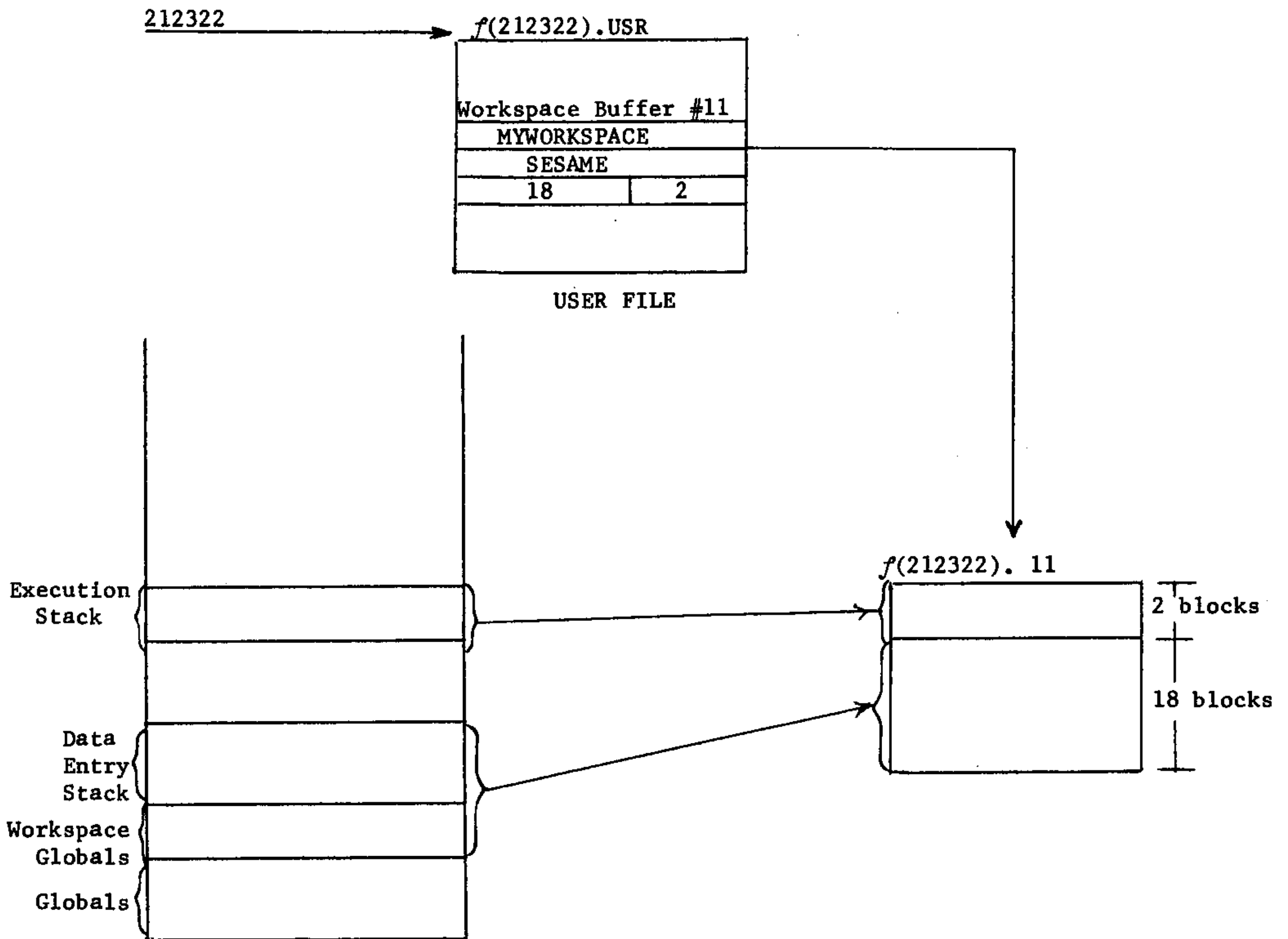
WORKSPACE BUFFER #20

USER FILE

buffer having a workspace name which matches the name given in the command or an empty workspace buffer depending on whether or not the workspace of the given name previously existed. If the nth workspace buffer is assigned to the workspace, the workspace is saved on the PDP-10 disk file <LIBRARY NO>.n. (Figure VII) For the )LOAD command, the system reads the appropriate user file into core, and searches the workspace buffers for the given workspace. The number of the workspace buffer determines what PDP-10 file contains the workspace. The workspace buffer also contains the information on how many blocks of the PDP-10 file to load into the low portion of the workspace and how many blocks to load into the high portion of the workspace.

FIGURE VII

)SAVE 212322 MYWORKSPACE: SESAME



BIBLIOGRAPHY

1. Abrams, P., An APL Machine, Stanford Linear Accelerator Center Report No. 114 (February 1970), Stanford, California.

2. Conway, M. E., "Design of a Separable Transition-Diagram Compiler", Communications of the ACM 6 (1963), 396.

3. Digital Equipment Corporation, PDP-10 Reference Handbook, 1969.

4. Falkoff, A. D. and K. E. Iverson, APL\360: User's Manual, International Business Machines Corporation, 1968.

5. Iverson, K. E., A Programming Language, John Wiley and Sons, New York, 1962.

6. Pakin, S., APL\360 Reference Manual, Science Research Associates, Inc., Chicago, 1968.

7. Wulf, W. A., D. Russell, A. N. Habermann, C. Geschke, J. Apperson, D. Wile, R. Brender, Bliss Reference Manual, Department of Computer Science document, Carnegie-Mellon University, Pittsburgh, Pa., 1970.

# APPENDIX A

## SYSTEM ADMINISTRATOR'S GUIDE

There are three classes of APL users: the APL operator, privileged users, and non-privileged users. There are two types of privileged user: permanently privileged and temporarily privileged. A permanently privileged user has privileges each time he signs on the system. A temporarily privileged user has privileges only from the time he is made privileged until he signs off.

Only the APL operator and privileged users may execute the following commands:

1. ADD:

)ADD <APL NO.> [<ID>] [:<PASSWORD>] <DISKQUOTA> <WSQUOTA> <N|P>

This command is used to join new users to the system. The <APL NO.> is any integer less than $2^{18}$. The optional <ID> is installation dependent. The password, if given, is an initial sign-on password. The disk quota is an integer followed by the letter K and is the maximum amount of disk space in K (1024 words) allotted for the storage of workspaces. The <WSQUOTA> is an integer establishing a limit on the number of workspaces the user may have. Only the APL operator may give the N or P parameter. The P specifies that the user is permanently privileged and N specifies no privileges. The default is N.

If the user has been joined to the system, the )ADD command may be used to modify his quotas, password, id, or privileges. In this case the disk and workspace quotas are added algebraically to the current quotas. (Note: APL numbers less than 1000 are public libraries.)

2. BOUNCE:

)BOUNCE <PORT NUMBER>

The BOUNCE command disconnects the user on the specified terminal from the system after saving his active workspace in the workspace named CONTINUE. No user may bounce the APL operator.

3. DELETE:

)DELETE <LIBRARY NUMBER>

The )DELETE command has the opposite effect of the )ADD command. It completely eliminates the library from the system. All workspaces under a library number must be dropped before the number can be deleted. No user may delete the APL operator.

4. LIB:

)LIB <LIBRARY NUMBER>

Unlike non-privileged users, privileged users may give a non-public library number with the )LIB command. In this case the system responds with a list of workspaces belonging to the given library number.

5. LOCK:

)LOCK <APL NUMBER>

The )LOCK command prohibits the user with the given APL number from signing on the system. The user's workspaces are in no way affected. No user may lock the APL operator.

6. PRIV:

)PRIV <PORT NUMBER>

The )PRIV command gives the privileged status to the user at the

given terminal. The user remains privileged only for the duration of his terminal session.

7. RESET:

)RESET <PORT NUMBER>

The )RESET command is used to recover an unused port buffer which the system erroneously thinks is in use. For example, if a user disconnects himself from the APL by typing ↑C (CONTROL C) and attempts to sign-on again, the system will give a 'NUMBER IN USE' error message. The )RESET command will then erase the erroneous information that user is signed-on, and subsequently allow the user to sign-on.

8. SAVE and DROP:

)SAVE <LIBRARY NUMBER> <WORKSPACE NAME>

)DROP <LIBRARY NUMBER> <WORKSPACE NAME>

Privileged users may specify any library number in the SAVE and DROP commands. In the case of the )SAVE command, the active workspace is saved with the given name under the given library number. Similarly with the )DROP command the named workspace is dropped from the given library number.

9. UNLOCK:

)UNLOCK <APL NUMBER>

The )UNLOCK command is the complement of the LOCK command. The )UNLOCK command removes the 'lock' placed upon the APL number and allows that user to sign-on.

# APPENDIX B

## PDP-10 APL PRELIMINARY MANUAL

From the user's viewpoint the PDP-10 APL is so similar to the IBM 360 APL that APL\360 User's Manual and APL\360 Reference Manual will serve as adequate references. However, some of the extensions, modifications, and omissions are:

### COMMANDS

Except as noted below, all commands work as in the above references.

1. CONTINUE and OFF:

    )CONTINUE [HOLD] and )OFF [HOLD]

    These commands all function as if the HOLD were present and return the user to monitor mode (after creating a CONTINUE workspace in the case of the CONTINUE command).

2. DISK:

    )DISK [<WS-NAME>]

    This command responds with the disk space required by each of the user's workspaces (no argument given) or by just the named workspace (argument given).

3. ECHO:

    )ECHO [ON|OFF]

    This command causes the error line and following caret line to be printed or suppressed when the parameter is ON or OFF respectively.

Without an argument the system responds with the current setting (defaulted to ON at sign-on).

4. MODE:

)MODE [KEYWORK|ESCAPE]

This command determines the teletype output mode. Without a parameter the system responds with the current mode (defaulted to KEYWORD at sign-on).

5. MON:

)MON

The user is returned to PDP-10 monitor mode. Typing the monitor command CONT will cause APL to resume.

6. SIZE:

)SIZE [<KSPEC>]

This command causes the size of the active workspace to be changed to the size specified. The size may not be decreased to less than 4K. Without an argument the system responds with the current size of the active workspace.

7. TIME:

)TIME

The system responds to this command with the amount of CONNECT and CPU time accumulated while the active workspace has been active.

8. WORKSPACE SIZE SPECIFICATION:

)CLEAR, )LOAD and SIGN-ON

A <K-SPEC> may be appended to these commands causing the new workspace to be of the specified size.

<K-SPEC> ::= <INTEGER>K

RESTRICTIONS

Under the current implementation, the )COPY command does not work for copying APL groups and the )PCOPY command does not print out a list of not-copied items.

TTY SYSTEM (See included transliteration tables)

APL\10 supports input from both 2741-type terminals and teletypes. The 2741 support requires specific monitor changes to the DEC scanner service package (which does not currently support 2741's). The teletype support requires no monitor modifications and assumes a single-case keyboard. For APL characters that do not appear directly on the teletype keyboard, a keyword/escape input system is provided (see below).

1. Two input modes (keyword and escape) are supported. For example, ρ may be input as '.RO' (keyword) or as '@R' (escape). No delimiting blanks are necessary, and the input modes may be mixed at will (for example, 3 4 .RO @I 12).

2. Two corresponding output modes are also supported. To select an output mode use the command

)MODE [KEYWORD|ESCAPE].

'KEYWORD' is the default mode.

3. <RUBOUT> is handled by the PDP-10 monitor as usual.

4. There is no way to input a <backspace>. Instead, there are keywords for all overstrikes.

5. '↑Q' currently serves as the attention signal for TTY's. Since this is a regular input character and is not handled at interrupt level, output may continue for a short while before the '↑Q' is detected. Be patient! Note: During function execution, if an attention is signaled, function execution is suspended and a message printed. Execution may be resumed by typing '→ <LINENUMBER>' (or '.GO <LINENUMBER>' for TTY's); however, execution resumes from the beginning of the interrupted line.

   Note: To suppress APL output on the teletype, '↑O' may be typed instead of '↑Q'. Since '↑O' is trapped by the monitor and works at an interrupt level, its action is immediate (instead of waiting for APL to handle a '↑Q' request). Of course, '↑Q' must still be used to interrupt function execution.

   On 2741-type terminals, the 'ATTN' key is used as usual; but, again, output may continue for a short while before APL processes the interrupt request (due to monitor queueing of printed output).

6. Warning: Since a TTY is full-duplex, typing ahead is physically possible; however, it is advised that it not be done. Unexpected cases may not be handled properly.

IMMEDIATE LINE EDITING

The last line typed in may be edited using the methods of function line editing. The edit request syntax is '[<ANY VALID LINENUMBER>⎕<COLUMN>]', and editing proceeds as if in function definition mode.

Note: If the next line typed is another edit request, the source line of the last edit is used again. (That is, the last edit request is not edited.)

ILLEGAL OVERSTRIKES

Typing an illegal overstrike will cause the remainder of the line to be ignored, and APL will prompt for a corrected line extension. For example:

```
        A ← B+C.NXD,E
    ?
        A ← B+C           (carriage waiting after 'C' for further
                           input)
```

FUNCTION DEFINITION

1. To delete line <N> from the currently open function, type

   '[∆<N>]' (or '[.LD <N>]' for TTY's).

2. Line labels are treated like local variables.

## QUAD MODE INPUT

Function definition is permitted during a quad input request. The input request remains open until an immediate line is typed in.

## PRIME-MODE INPUT

A new input mode similar to the quote-quad mode is supported. A prime-mode input request is made by typing '⍢' (or '.QD' for TTY's). APL then accepts an input line (delimited by a <CR>) and creates a character string from it. In prime-mode input, no editing is done on the input line. That is, for 2741's, AB<BS>_C goes through to those individual characters (and the overstrike 'B̲' is not created); and for TTY's, 5.RO@I5 will go through to the corresponding seven character string. This mode is useful in developing text-editing systems.

# MNEMONICS FOR APL\10 TTY SYSTEM

| | Mnemonic | APL character | Alternate 'mnemonic' |
|---|---|---|---|
| | .AL | α | @A |
| | .DE | ⊥ | @B |
| | .DU | ∩ | @C |
| | .FL | ⌊ | @D |
| | .EP | ∊ | @E |
| | .US | _ | @F |
| | .DL | ∇ | @G |
| | .LD | ∆ | @H |
| | .IO | ⍳ | @I |
| | .SO | ∘ (null) | @J |
| | ' | ' | @K |
| | .BX | ⎕ | @L |
| | .AB | \| | @M |
| | .EN | ⊤ | @N |
| | .LO | ○ | @O |
| | * | ⋆ | @P |
| | ? | ? | @Q |
| | .RO | ⍴ | @R |
| | .CE | ⌈ | @S |
| | .NT or $ | ∼ | @T |
| | .DA | ↓ | @U |
| | .UU | ∪ | @V |
| | .OM | ⍵ | @W |
| | .LU | ⊃ | @X |
| | ↑ | ↑ | @Y |
| | .RU | ⊂ | @Z |
| Overstrikes: | .CB | ⍀ | |
| | .CR | ⊖ | |
| | .CS | ⌿ | |
| | .GD | ⍒ | |
| | .GU | ⍋ | |
| | .IB | ⌶ | |
| | .LG | ⍟ | |
| | .NN | ⍲ | |
| | .NR | ⍱ | |
| | .OU | *O <BS> U <BS> T* | |
| | .PD | ⍫ | |
| | .QD | ⌹ | |
| | .QQ | ⍞ | |
| | .RV | ⌽ | |
| | .TR | ⍉ | |
| | .XQ | ⍎ | |
| | .ZA | <u>*A*</u> | |
| | .ZB, etc. | <u>*B*</u>, etc. | |

Necessary mnemonics:

| | |
|---|---|
| .DA | ↓ |
| .DD | ¨ |
| .GE | ≥ |
| .GO | → |
| .LE | ≤ |
| .NE | ≠ |
| .NG | ¯ |
| .OR | ∨ |

Also:

| TTY | APL |
|---|---|
| " | ⍝ |
| & | ∧ |
| # | × |
| % | ÷ |
| ! | ! |
| ( | ( |
| ), etc. | ), |

# APPENDIX C

## OBTAINING AND MOUNTING THE APL SYSTEM

Tapes containing copies of source files, a Bliss compiler binary file, and the APL system binary file are available from the Computer Science Department, Carnegie-Mellon University. These files are available on Magtape (stored on magtape by the SAVE cusp) or on Dectapes (stored on Dectape by the PIP cusp). There is a nominal charge for the Magtape copy and the Dectape copy to cover the costs of tapes, duplication, and postage.

### MOUNTING THE APL SYSTEM

1. Obtain a project-programmer number under which all user and workspace files are to be stored.

2. Move all files to disk from the Magnetic tape using the SAVE cusp or from Dectapes using the PIP cusp.

3. Store the files DIRECTORY AND Ø5YA.USR under the PPN obtained in 1.

4. Perform the following monitor commands:

   . GET DSK APL

   . D <PROJ. NO.> <PROG. NO.> 400013; — Here <PROJ. NO.> <PROG. NO.> are the PROJECT and PROGRAMMER NUMBER of 1 above.

   . SAVE DSK <ANY FILENAME>

5. The file created by the last save command is the APL system which may be made a cusp. Use the RUN (or R) monitor command to start execution of the APL system. When the terminal spaces over 6 spaces type:

)6400 <CR>

The system will respond with a greeting message. The APL operator is then signed on and is free to add additional users. (See the ADD command.)

Security Classification

# DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

| 1. ORIGINATING ACTIVITY (Corporate author) | 2a. REPORT SECURITY CLASSIFICATION |
|---|---|
| Department of Computer Science<br>Carnegie-Mellon University<br>Pittsburgh, Pennsylvania 15213 | UNCLASSIFIED |
| | 2b. GROUP |

3. REPORT TITLE

Conversational Programming--APL an Implementation in Bliss

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Scientific Interim

5. AUTHOR(S) (First name, middle initial, last name)

A. J. Perlis, R. D. Fennell, F. J. Pollack, W. R. Price, M. F. Rizzo

| 6. REPORT DATE | 7a. TOTAL NO. OF PAGES | 7b. NO. OF REFS |
|---|---|---|
| June 1971 | 52 | 7 |

| 8a. CONTRACT OR GRANT NO. | 9a. ORIGINATOR'S REPORT NUMBER(S) |
|---|---|
| F44620-70-C-0107 | |
| b. PROJECT NO. A0827-5 | |
| c. 61101D | 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) |
| d. | |

10. DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

| 11. SUPPLEMENTARY NOTES | 12. SPONSORING MILITARY ACTIVITY |
|---|---|
| TECH, OTHER | Air Force Office of Scientific Research<br>1400 Wilson Boulevard<br>Arlington, Virginia 22209 |

13. ABSTRACT

As part of the ongoing research program in conversational programming an APL system has been implemented for the PDP-10. Since this system is to be a base for extensive study in conversational programming the system was programmed entirely in Bliss, a high-level programming language specifically designed for the writing of systems programs. A few extensions to APL are included in this first version which supports both Teletype and IBM/Datel terminals.

DD FORM 1 NOV 65 1473

Security Classification

Security Classification

| 14. KEY WORDS | LINK A | | LINK B | | LINK C | |
|---|---|---|---|---|---|---|
| | ROLE | WT | ROLE | WT | ROLE | WT |
| | | | | | | |

Security Classification